Figure 3

Do not forget to cite that we extended the Impacted Class implementation and create this new sophisticated technique for evaluating SPL evolutions.

In order to solve this problem, we propose an approach entitled Extended Impacted Class, which goes beyond of focusing on changed classes without taking into account all the contexts.

As you can see in figure 3, instead of checking the impacted class, we are interest on the backward impacted classes.

Instead of checking the impacted class, we are interested on the backward dependencies, because the local behavioral change might be camouflaged on the superior classes. As you can see on the motivating example, we disconsider the *TestSuite* Class where the change has been applied and now our focus is on the most class close to the Graphical User interface, because as closer we are of the user, more we are accurate in the process of checking refinement.

Note that, when we perform a code analysis on the *TestSuiteComparison* class, we perceive that the apparently bad code transformation do not affect the product line users since it ensures that the object which invokes the comparison method will never be null as explained before. Therefore, we are more confident on the EIC approach which states that this evolution is a refinement. This strategy avoids false-negatives.

* How we find the backward impacted classes.

To find the backward impacted classes we extend the soot framework

* How we know these classes are below the Graphical User Interface level.

You might be wondering, how we know these classes are below the Graphical User Interface level. Our technique has been designed to receive as a parameter the GUI component because our tool set do not support test generation for these elements. Graphical User Interface testing is far more difficult than the conventional software testing applied in model classes. Many new and complex problems arise in the process. A classic challenge in GUI testing is how to select a feasible number of event sequences, given the combinatorial explosion due to arbitrary event interleaving. To illustrate, consider testing a GUI with one textbox that takes a ten character string as an input. Exhaustive testing requires 10^26 possible input strings. In this context, combinatorial explosion might be a problem when the software deals with a lot of branching and choices. In such scenarios performing GUI testing can be extremely difficult. To overcome this problem, we are evaluating this strategy of exclusively test one level below the GUI.

We are considering a new test generation tools integration to our toolset in order to overcome this

To overcome this problem, we are evaluating this strategy of exclusively test one level below the GUI.

We are evaluating this strategy of exclusively test one level below the GUI, however, we are considering to integrate to the toolset a new test generation tools